Declarative Output by Ordering Text Pieces

Stefan Brass
University of Halle, Germany

Motivation (1)

Is declarative output necessary?

- Declarativity is an important principle of logic programming and of databases.
- Among other things, it simplifies optimizations.
- Maybe declarative output is not essential if
  - ◇ 99% of the program is a complex calculation and
  - ◇ only a very small part of the program outputs the result at the end.
- But database applications are different: Here output is a large and important part.

Motivation (2)

Aren't there good solutions?

- A typical declarative solution is to use an IO state argument (accumulator pair):

```
main(IOState_in, IOState_out) :-
    io.write_string("Hello, ", IOState_in,
                               IOState_1),
    io.write_string("World!",  IOState_1,
                               IOState_2),
    io.nl(IOState_2, IOState_out).
```

This is from the Mecury Tutorial. Mecury has a special syntax that simplyfies this a bit, and also checks for determinismn (one cannot backtrack over IO).

Motivation (3)

Problems with the standard solution:

- Even printing a simple table represented as facts needs `findall`, lists, and recursion.
- It does not fit the bottom-up way of thinking about rules (from right to left, direction of $\leftarrow$), usual in deductive databases.
- The program must be written in a way that backtracking over output can never occur.

  The programmer has to assure this, maybe in a way that the system can check it.

A Simple Solution (1)

- Output is done by deriving facts about a predicate "`output`" with two arguments:
  - ◇ The first argument defines the output position.
  - ◇ The second one is a piece of the generated text.
- The derived facts are sorted by the first argument, then the second argument is printed in that order.

```
output(1, "Hello, ").
output(2, Name) ← user(Name).
output(3, "!\n").
user("Nina").
```

A Simple Solution (2)

- In order to support hierarchically structured documents, lists are permitted as first argument.

  With the standard lexicographic order, i.e. the list first element, which is not considered equivalent (such as two strings that differ only in lower/uppercase) decides. In the paper, nested lists and other terms are not considered, but that might open interesting possibilities (e.g. terms that order their arguments backwards).

- By using data values inside the list, one gets the possibility to sort the text pieces by data values.

  One can have several sorting criteria of different priority.

A Simple Solution (3)

Homework Results as HTML ordered by Points, Name:

```
output([1], "<table>").
output([2], "<tr><th>Points</th><th>Name</th></tr>").
output([3,P,N,1], "<tr><td>")    ← homework(N, P).
output([3,P,N,2], P)             ← homework(N, P).
output([3,P,N,3], "</td><td>")   ← homework(N, P).
output([3,P,N,4], N)             ← homework(N, P).
output([3,P,N,5], "</td></tr>") ← homework(N, P).
output([4], "</table>").

homework("Ann", 5).
homework("Bob", 10).
homework("Chris", 10).
```

Important Notice

- While the above approach can be used to give a declarative semantics to output, and theoretically investigate the possibilities,
  - ◇ the syntax is too complicated for actual usage,
  - ◇ sorting is a relatively expensive operation, so a direct implementation might be inefficient.
- Both problems can be solved.

Syntax Improvement (1)

- New syntax with "output patterns":

```
homeworks_table(#
    <table>
    <tr><th>Points</th><th>Student</th></tr>
    <#homeworks_row>
    </table>
#).
```

- This is automatically translated to:

```
homeworks_table([1], "<table>\n").
homeworks_table([2], "<tr><th>Points</th>...").
homeworks_table([3|X], Y) ← homeworks_row(X, Y).
homeworks_table([4], "</table>").
```

Syntax Improvement (2)

- Second part (table rows, sorted):

```
homeworks_row([Points, Name]#
    <tr><td><$Points></td><td><$Name></td></tr>
#) ← homeworks(Name, Points).
```

- This makes it possible to instantiate a pattern several times with different data values and specify the order of the result. Corresponding rules:

```
homeworks_row([Points, Name, 1], "<tr><td>").
homeworks_row([Points, Name, 2], Points).
homeworks_row([Points, Name, 3], "</td><td>").
homeworks_row([Points, Name, 4], Name).
homeworks_row([Points, Name, 5], "</td></tr>").
```

Syntax Improvement (3)

- Of course, pattern syntax can also be used for the predicate "`output`".
- This is the "main" predicate, which defines the entire output / web page generated by the program:

```
output(#
    <html>
    <head><title>Homework Results</title></head>
    <body><h1>Homework Results</h1>
    <#homeworks_table>
    </body></html>
#).
```

Efficient Evaluation (1)

- A tuple stream (cursor/iterator) interface for predicates/relations is used.

  As usual, we try not to materialize intermediate relations in order to save memory (unless the same relation is used several times and it is cheaper to store it than to recompute it, or we need explicit sorting).

- A tuple stream can be ordered by given arguments.
- I.e. the task is to translate the given rules for a predicate into e.g. C++-code for iterators that produce the tuples in the required order.

  In the end, one wants the `output`-tuples ordered by first argument, but it helps if the predicates used in the `output`-rules can be fetched in a specific order.

Efficient Evaluation (2)

- For several rules about one predicate, it is often obvious that all tuples produced by one rule come before all tuples produced by another rule.

```
homeworks_table([1], "<table>\n").
homeworks_table([2], "<tr><th>Points</th>...").
homeworks_table([3|X], Y) ← homeworks_row(X, Y).
homeworks_table([4], "</table>").
```

- So we choose the rule evaluation sequence correspondingly (avoids work at runtime).
- Otherwise ordered tuple streams for the single rules can be efficiently merged.

Efficient Evaluation (3)

- Also the nested loop join preserves certain orders.

  The full version of the paper contains an unconventional variant that does the sorting in groups, and helps to preserve more order.

- Base tables might have indexes or be stored in sorted order (e.g., as a b-tree), then tuples can be accessed in certain sorted orders.

- Sometimes, explicit sorting is unavoidable, and it is an optimization task to find the optimal place(s).

  It does not necessarily have to be at the very end. Sorting also helps with merge joins.

Conclusion

- For me, the main promise of deductive databases is that a single declarative language is used for
  - ◇ database queries, and
  - ◇ all usual programming tasks including output.

    Maybe seldom additional built-in predicates can be written in another language like C++.
- Our goal is to write a deductive database system mainly in Datalog, and to do this by a transformation from Datalog to C++.

  Thus output (for the generated code) is important for the system implementation itself.